目錄

React Bits 中文版

有关React，你需要知道的一切

Gitbook format: https://wizardforcel.gitbooks.io/react-bits

Repo 地址 https://github.com/hateonion/react-bits-CN

> Your contributions are heartily ♡ welcome. (✿◠‿◠) 新司机上路，欢迎大家随时提issue和pr进行指正

JSX中的状态分支

在以下情况下，和使用三元运算符相比

```
const sampleComponent = () => {
  return isTrue ? <p>True!</p> : <none/>
};
```

使用&&符号的表达式简写会是一种更好的实践

```
const sampleComponent = () => {
  return isTrue && <p>True!</p>
};
```

如果像下面一样有很多的三元运算符的话:

```
// Y soo many ternary??? :-/
const sampleComponent = () => {
  return (
    <div>
      {flag && flag2 && !flag3
        ? flag4
        ? <p>Blah</p>
        : flag5
        ? <p>Meh</p>
        : <p>Herp</p>
        : <p>Derp</p>
      }
    </div>
  )
};
```

- 最佳实践: 将逻辑移到子组件内部
- 另一种hack的做法: 使用IIFE(Immediately-Invoked Function Expression 立即执行函数)

有一些库可以解决用jsx控制组件状态的问题，但是这些外部依赖并不是必须的，我们可以使用 IIFE 将if-else的逻辑封装到组件内部，外部调用者并不需要关心这些逻辑，正常调用即可。

```
const sampleComponent = () => {
  return (
    <div>
      {
        (() => {
          if (flag && flag2 && !flag3) {
            if (flag4) {
              return <p>Blah</p>
            } else if (flag5) {
              return <p>Meh</p>
            } else {
              return <p>Herp</p>
            }
          } else {
            return <p>Derp</p>
          }
        })()
      }
    </div>
  )
};
```

或者也可以在满足条件的时候使用return强制跳出函数，这样能避免后面冗余的判断执行。

```
const sampleComponent = () => {
  const basicCondition = flag && flag2 && !flag3;
  if (!basicCondition) return <p>Derp</p>;
  if (flag4) return <p>Blah</p>;
  if (flag5) return <p>Meh</p>;
  return <p>Herp</p>
}
```

参考资料:

- https://engineering.musefind.com/our-best-practices-for-writing-react-components-dec3eb5c3fc8
- Conditional rendering

setState函数的异步性

简述

在某些情况下，React框架出于性能优化考虑，可能会将多次state更新合并成一次更新。正因为如此，setState实际上是一个异步的函数。 但是，有一些行为也会阻止React框架本身对于多次state更新的合并，从而让state的更新变得同步化。 比如: eventListeners, Ajax, setTimeout 等等。

详解

当setState() 函数执行的时候，函数会创建一个暂态的state作为过渡state，而不是立即修改this.state。 如果在调用setState()函数之后尝试去访问this.state，你得到的可能还是setState()函数执行之前的结果。 在使用setState()的情况下，看起来同步执行的代码其实执行顺序是得不到保证的。原因上面也提到过，React可能会将多次state更新合并成一次更新来优化性能。

运行下面这段代码，你会发现当和addEventListener, setTimeout 函数或者发出AJAX call的时候，调用setState, state会发生改变。并且render函数会在setState()函数被触发之后马上被调用。那么到底发生了什么呢？事实上，类似setTimeout()函数或者发出ajax call的fetch函数属于调用浏览器层面的API，这些函数的执行并不存在与React的上下文中，所以React并不能够像控制其他存在与其上下文中的函数一样，将多次state更新合并成一次。

在上面这些例子中，React框架之所以在选择在调用setState函数之后立即更新state而不是采用框架默认的方式，即合并多次state更新为一次更新，是因为这些函数调用(fetch,setTimeout等浏览器层面的API调用)并不处于React框架的上下文中，React没有办法对其进行控制。React在此时采用的策略就是及时更新，确保在这些函数执行之后的其他代码能拿到正确的数据（即更新过的state)。

```
class TestComponent extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
```

```
      dollars: 10
    }
    this.onMouseLeaveHandler = this.onMouseLeaveHandler.bind(this
);
    this.onTimeoutHandler = this.onTimeoutHandler.bind(this);
    this.onAjaxCallback = this.onAjaxCallback.bind(this);
    this.onClickHandler = this.onClickHandler.bind(this);
  }

  componentDidMount() {
    // Add custom event via `addEventListener`
    //
    // The list of supported React events does include `mouselea
ve`
    // via `onMouseLeave` prop
    //
    // However, we are not adding the event the `React way` - th
is will have
    // effects on how state mutates
    //
    // Check the list here - https://facebook.github.io/react/do
cs/events.html
    document.getElementById('testButton').addEventListener('mous
eleave', this.onMouseLeaveHandler);

    // Add JS timeout
    //
    // Again,outside React `world` - this will also have effects
 on how state
    // mutates
    setTimeout(this.onTimeoutHandler, 10000);

    // Make AJAX request
    fetch('https://api.github.com/users')
      .then(this.onAjaxCallback);
  }

  onClickHandler = () => {
    console.log('State before (_onClickHandler): ' + JSON.string
ify(this.state));
```

```
    this.setState({
      dollars: this.state.dollars + 10
    });
    console.log('State after (_onClickHandler): ' + JSON.stringi
fy(this.state));
  }

  onMouseLeaveHandler = () => {
    console.log('State before (mouseleave): ' + JSON.stringify(t
his.state));
    this.setState({
      dollars: this.state.dollars + 20
    });
    console.log('State after (mouseleave): ' + JSON.stringify(th
is.state));
  }

  onTimeoutHandler = () => {
    console.log('State before (timeout): ' + JSON.stringify(this
.state));
    this.setState({
      dollars: this.state.dollars + 30
    });
    console.log('State after (timeout): ' + JSON.stringify(this.
state));
  }

  onAjaxCallback = (err, res) => {
    if (err) {
      console.log('Error in AJAX call: ' + JSON.stringify(err));
      return;
    }

    console.log('State before (AJAX call): ' + JSON.stringify(th
is.state));
    this.setState({
      dollars: this.state.dollars + 40
    });
    console.log('State after (AJAX call): ' + JSON.stringify(this
.state));
```

```
  }

  render() {
    console.log('State in render: ' + JSON.stringify(this.state)
);

    return (
      <button
        id="testButton"
        onClick={this.onClickHandler}>
        'Click me'
      </button>
    );
  }
}

ReactDOM.render(
  <TestComponent />,
  document.getElementById('app')
);
```

解决setState函数异步的办法?

根据React官方文档，setState函数实际上接收两个参数，其中第二个参数类型是一个函数，作为setState函数执行后的回调。通过传入回调函数的方式，React可以保证传入的回调函数一定是在setState成功更新this.state之后再执行。

例子

```
_onClickHandler: function _onClickHandler() {
   console.log('State before (_onClickHandler): ' + JSON.stringi
fy(this.state));
   this.setState({
   dollars: this.state.dollars + 10
   }, () => {
   console.log('Here state will always be updated to latest vers
ion!');
   console.log('State after (_onClickHandler): ' + JSON.stringif
y(this.state));
   });
}
```

更多关于setState的小知识

其实setState作为一个函数，本身是同步的。只是因为在setState的内部实现中，使用了React updater的enqueueState 或者 enqueueCallback方法，才造成了异步。

下面这段是React源码中setState的实现:

```
ReactComponent.prototype.setState = function(partialState, callb
ack) {
  invariant(
    typeof partialState === 'object' ||
    typeof partialState === 'function' ||
    partialState == null,
    'setState(...): takes an object of state variables to update
 or a ' +
    'function which returns an object of state variables.'
  );
  this.updater.enqueueSetState(this, partialState);
  if (callback) {
    this.updater.enqueueCallback(this, callback, 'setState');
  }
};
```

而updater的这两个方法，又和React底层的Virtual Dom(虚拟DOM树)的diff算法有紧密的关系，所以真正决定同步还是异步的其实是Virtual DOM的diff算法。

参考资料:

- https://medium.com/@wereHamster/beware-react-setstate-is-asynchronous-ce87ef1a9cf3#.jhdhncws3
- https://www.bennadel.com/blog/2893-setstate-state-mutation-operation-may-be-synchronous-in-reactjs.htm

依赖注入

在React中，想做依赖注入(Dependency Injection)其实相当简单。请看下面这个例子:

```
// Title.jsx
export default function Title(props) {
  return <h1>{ props.title }</h1>;
}
```

```
// Header.jsx
import Title from './Title.jsx';
export default function Header() {
  return (
    <header>
      <Title />
    </header>
  );
}
```

```
// App.jsx
import Header from './Header.jsx';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { title: 'React Dependency Injection' };
  }
  render() {
    return <Header />;
  }
}
```

试想，我们想把"React Dependency Injection"这个字符串传到我们的Title组件里面去。比较直观的方法是把这个字符串作为props传入Header组件，然后Header组件将这个字符串作为props传入Title组件。 在只有三个组件相互嵌套的情况下，上面

的这种解决方式似乎能满足我们的需要。但是设想一下，随着组件数量的增加以及组件嵌套层次的加深，许多的组件都需要接收该组件本身并不需要关心的props，然后且简单地传递给子组件。直观上来看，上面的这种方法在这种情况下，似乎就不太适用了。

不过不用担心，事实上我们还有很多方法能实现依赖的注入，其中之一就是高阶组件(high-order component)。

```
// inject.jsx
var title = 'React Dependency Injection';
export default function inject(Component) {
  return class Injector extends React.Component {
    render() {
      return (
        <Component
          {...this.state}
          {...this.props}
          title={ title }
        />
      )
    }
  };
}
```

```
// Title.jsx
export default function Title(props) {
  return <h1>{ props.title }</h1>;
}
```

```
// Header.jsx
import inject from './inject.jsx';
import Title from './Title.jsx';

var EnhancedTitle = inject(Title);
export default function Header() {
  return (
    <header>
      <EnhancedTitle />
    </header>
  );
}
```

在上面这种实现中，title这个字符串作为我们的数据没有被注入到整个的组件树中。相反，我们的数据只传递给了需要关注这个数据的组件。 这种实现看上去比上面优雅了很多，但事实上并没有完全解决我们的问题。 我们将title这个字符串通过inject这个高阶组件注入了进去，但是如果我的title字符串只能在我的Header组件这个层级获得呢？那么是不是意味着我还需要再从Header组件传递给inject组件呢？

这是一个实际可能会发生的问题，为了解决这个问题，接下来需要介绍一下React的Context API。 在React框架中，存在着context这样一个概念。context有点类似与事件总线(Event Bus)，是一个无论在哪都可以访问的对象。不同的是，但是在context里面维持的全部都是我们的数据而非事件。context贯穿于整个组件树中，所有的组件都可以访问context。

使用方法

```
var context = { title: 'React in patterns' };
class App extends React.Component {
  getChildContext() {
    return context;
  }
  // ...
}

App.childContextTypes = {
  title: PropTypes.string
};
```

A place where we need data

```
class Inject extends React.Component {
  render() {
    var title = this.context.title;
  // ...
  }
}
Inject.contextTypes = {
  title: PropTypes.string
};
```

需要注意的是，在最新的React官方文档中，Context已经不太被官方推荐使用了。Why Not To Use Context

参考资料:

- What is Dependency Injection?
- The Basics of Dependency Injection
- Dependency injection in JavaScript
- DI In React

包裹Context

相比于单纯的数据对象，将context包装成一个提供一些方法的对象会是更好的实践。因为这样能提供一些方法供我们操作context里面的数据。

```
// dependencies.js
export default {
  data: {},
  get(key) {
    return this.data[key];
  },
  register(key, value) {
    this.data[key] = value;
  }
}
```

经过了包装的Context，可以通过类似于下面的这种方法使用。

```
import dependencies from './dependencies';
dependencies.register('title', 'React in patterns');

class App extends React.Component {
  getChildContext() {
    return dependencies;
  }
  render() {
    return <Header />;
  }
}

// 我们还可以对context中的数据做类型校验
App.childContextTypes = {
  data: PropTypes.object,
  get: PropTypes.func,
  register: PropTypes.func
};
```

这样我们的Title组件就能直接从Context中获取数据了。

```
// Title.jsx
export default class Title extends React.Component {
  render() {
    return <h1>{ this.context.get('title') }</h1>
  }
}
Title.contextTypes = {
  data: PropTypes.object,
  get: PropTypes.func,
  register: PropTypes.func
};
```

一般来说，我们不需要每次在使用context的地方都对context内的数据做类型校验。这种功能完全可以借由一个高阶组件派生出来。我们甚至可以使用高阶组件来作为我们操作context的代理，来替代我们对于context的直接操作。

比如：我们可以使用一个高阶组件来替代我们直接对于this.context.get('title')方法的调用。

```
// Title.jsx
import wire from './wire';

function Title(props) {
  return <h1>{ props.title }</h1>;
}

export default wire(Title, ['title'], function resolve(title) {
  return { title };
});
```

wire这个函数的接收一个React Element作为第一个参数。第二个参数为一个数组，数组内容为组件所依赖的数据在context中的key。第三个参数我把它叫做mapper，mapper这个函数会将context里面的原始数据进行处理，然后以对象的形式返回组件所需要的数据。在我们的Title组件的例子中，我们在只需要在第二个参数中传入title作为我们依赖的描述，mapper就会返回在context中的title。 但是在实际应用中，我们所需要的数据可能会很多，依赖的描述形式也可能千变万化，可能是一堆

数据的集合，也可能是读自某个配置文件。但是我们都可以通过将依赖的描述传入wire函数，让wire函数去帮我们将所有必要的依赖传入我们的组件，而不是传入所有的context。

下面是一种可能的实现：

```
export default function wire(Component, dependencies, mapper) {
  class Inject extends React.Component {
    render() {
      var resolved = dependencies.map(this.context.get.bind(this
.context));
      var props = mapper(...resolved);

      return React.createElement(Component, props);
    }
  }
  Inject.contextTypes = {
    data: PropTypes.object,
    get: PropTypes.func,
    register: PropTypes.func
  };
  return Inject;
};
```

Inject 是一个能访问context并且获取所有在dependency数组中列出的数据的高阶组件。mapper是一个能接受context数据作为输入，将所有需要的数据从context中取出转换成组件props的函数。

不依赖context的另外一种实现

我们使用一个单例来注册/获取所有的依赖

```
var dependencies = {};

export function register(key, dependency) {
  dependencies[key] = dependency;
}

export function fetch(key) {
  if (key in dependencies) return dependencies[key];
  throw new Error(`"${ key } is not registered as dependency.`);
}

export function wire(Component, deps, mapper) {
  return class Injector extends React.Component {
    constructor(props) {
      super(props);
      this._resolvedDependencies = mapper(...deps.map(fetch));
    }
    render() {
      return (
        <Component
          {...this.state}
          {...this.props}
          {...this._resolvedDependencies}
        />
      );
    }
  };
}
```

我们把dependencies这个对象存放在全局范围（不是应用全局范围，而是包全局范围）。同时我们export出register和fetch两个函数用于读写我们的dependencies对象。（有点类似javascript class里面getter和setter的实现）。至此，wire函数的实现就已经完成了。wire函数接受一个React组件作为输入，输出一个高阶组件。

我们在这个返回的高阶组件中去处理我们的依赖并将其转化为props,在render函数中传入子组件（即我们传入想真正渲染的组件）

遵循上面这个模式，我们实现了以下代码。di.jsx这个helper帮助我们将我们应用的所有以来注册好，并且通过这个helper我们可以在整个应用的域里面随时取得我们想需要的依赖。

```
// app.jsx
import Header from './Header.jsx';
import { register } from './di.jsx';

register('my-awesome-title', 'React in patterns');

class App extends React.Component {
  render() {
    return <Header />;
  }
}
```

```
// Header.jsx
import Title from './Title.jsx';

export default function Header() {
  return (
    <header>
      <Title />
    </header>
  );
}
```

```
// Title.jsx
import { wire } from './di.jsx';

var Title = function(props) {
  return <h1>{ props.title }</h1>;
};

export default wire(Title, ['my-awesome-title'], title => ({ tit
le }));
```

如果我们仔细观察 `Title.jsx` 的话，我们会发现实际上用到的component和wiring后的component的可以来自于不同的文件，这样的话，所有的这些代码都是可以被很容易的测试的。（因为可以很容易被mock）

事件处理

我们需要在constructor中对于事件与对应的handler函数进行绑定.

大多数时候我们在发出DOM事件的组件内部写我们的handler函数. 在下面的例子中,我们在组件内部创建了一个click handler, 因为我们想所有的Swithcer Component当被点击时,做出同样的响应.

```
class Switcher extends React.Component {
  render() {
    return (
      <button onClick={ this._handleButtonClick }>
        click me
      </button>
    );
  }
  _handleButtonClick() {
    console.log('Button is clicked');
  }
}
```

上面这样做完全没有问题,因为 `_handleButtonClick` 是一个函数, 我们把这个函数和onClick这个React支持的event绑定在了一起.

但是上面这样做也会带来问题, 使用function的写法, 会在function初始化时生成一个this. 比如我们在 `_handleButtonClick` 里面使用 `this` , 此时的this是 `_handleButtonClick` 生成出来的, 和Switcher这个class的this没有任何关系, 如果我想访问类似于this.props 或者 this.state这样的对象, 代码便会报错.

```
class Switcher extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'React in patterns' };
  }
  render() {
    return (
      <button onClick={ this._handleButtonClick }>
        click me
      </button>
    );
  }

  _handleButtonClick() {
    console.log(`Button is clicked inside ${ this.state.name }`)
;
    // 将导致
    // Uncaught TypeError: Cannot read property 'state' of null
  }
}
```

所以我们常用的解决办法像下面一样使用 `bind`

```
<button onClick={ this._handleButtonClick.bind(this) }>
  click me
</button>
```

然而，这种写法意味这我们要一次又一次的去调bind函数，因为我们的button可能会被渲染很多次。一种更好的做法是在组件的constructor中去做我们bind函数的调用。

```
class Switcher extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'React in patterns' };
    this._buttonClick = this._handleButtonClick.bind(this);
  }
  render() {
    return (
      <button onClick={ this._buttonClick }>
        click me
      </button>
    );
  }
  _handleButtonClick() {
    console.log(`Button is clicked inside ${ this.state.name }`)
;
  }
}
```

另一种办法是使用箭头函数建立我们的handler函数，因为箭头函数并不会创建 `this` .

顺带一提的是，Facebook也推荐使用这种方法去处理需要访问组件的 `this` 的函数.

但是，在constructor中去做binding也同样有用处. 比如，我们可能会将父组件中定义的函数作为Props传下去. 因此在子组件中，我们需要对这个函数进行bind.

Flux模式

简单的**dispatcher**实现

```
var Dispatcher = function () {
  return {
    _stores: [],
    register: function (store) {
      this._stores.push({store: store});
    },
    dispatch: function (action) {
      if (this._stores.length > 0) {
        this._stores.forEach(function (entry) {
          entry.store.update(action);
        });
      }
    }
  }
};
```

我们期望我们的store能提供一个update方法. 接下来让我们来修改一下我们的register函数.

```
function register(store) {
  if (!store || !store.update && typeof store.update === 'functi
on') {
    throw new Error('You should provide a store that has an upda
te method');
  } else {
    this._stores.push({store: store});
  }
}
```

完整的**dispatcher**实现

```
var Dispatcher = function () {
```

```
  return {
    _stores: [],
    register: function (store) {
      if (!store || !store.update && typeof store.update === 'fu
nction') {
        throw new Error('You should provide a store that has an
`update` method.');
      } else {
        var consumers = [];
        var change = function () {
          consumers.forEach(function (l) {
            l(store);
          });
        };
        var subscribe = function (consumer, noInit) {
          consumers.push(consumer);
          !noInit ? consumer(store) : null;
        };

        this._stores.push({store: store, change: change});
        return subscribe;
      }
      return false;
    },
    dispatch: function (action) {
      if (this._stores.length > 0) {
        this._stores.forEach(function (entry) {
          entry.store.update(action, entry.change);
        });
      }
    }
  }
};

module.exports = {
  create: function () {
    var dispatcher = Dispatcher();

    return {
      createAction: function (type) {
```

```
        if (!type) {
          throw new Error('Please, provide action\'s type.');
        } else {
          return function (payload) {
            return dispatcher.dispatch({type: type, payload: pay
load});
          }
        }
      },
      createSubscriber: function (store) {
        return dispatcher.register(store);
      }
    }
  }
};
```

参考资料:

- https://github.com/krasimir/react-in-patterns/tree/master/patterns/flux

单向数据流

单向数据流只关注于在store中维护的唯一的state,消除了不必要的多种states带来的复杂度. 这个store应该具有能被我们订阅(subscribe)store中变化的能力,实现如下:

```
var Store = {
  _handlers: [],
  _flag: '',
  onChange: function (handler) {
    this._handlers.push(handler);
  },
  set: function (value) {
    this._flag = value;
    this._handlers.forEach(handler => handler())
  },
  get: function () {
    return this._flag;
  }
};
```

然后我们会在我们的App组件上加上订阅我们的store的钩子, 当每次store发生改变的时候, 我们的组件就会被重新的渲染.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    Store.onChange(this.forceUpdate.bind(this));
  }

  render() {
    return (
      <div>
        <Switcher
          value={ Store.get() }
          onChange={ Store.set.bind(Store) }/>
      </div>
    );
  }
}
```

请注意我们使用了forceUpdate这个函数,但是事实上我们并不推荐使用这个函数.

一般情况下我们会使用高阶组件去帮我们处理重新渲染的事情,我们在这里使用forceUpdate函数只是想尽可能的保持我们这个例子简单.

因为我们使用了store, Swticher这个组件就变得超级简单了. 我们不需要再在Switcher组件内部再去维护一份State了:

```
class Switcher extends React.Component {
  constructor(props) {
    super(props);
    this._onButtonClick = e => {
      this.props.onChange(!this.props.value);
    }
  }

  render() {
    return (
      <button onClick={ this._onButtonClick }>
        { this.props.value ? 'lights on' : 'lights off' }
      </button>
    );
  }
}
```

使用单向数据流的好处是我们的组件在这种情况下变得非常声明式, 成为了所谓的纯UI组件, 只是我们Store里面数据的表达. React在诞生之初就是为了解决视图层(View层)的问题, 其核心思想也是从视图出发去解决问题. 我们用声明式的方式去构建我们的应用, 让我们的组件变得尽可能的木偶化,只关心我们的数据,将我们的复杂的数据逻辑交由我们的store去处理. 这也是我们去构建React应用的时候的一种比较好的思路.

参考资料:

- https://www.startuprocket.com/articles/evolution-toward-one-way-data-flow-a-quick-introduction-to-redux

展示组件和容器组件

问题描述

UI和业务逻辑和数据混杂在一起.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {time: this.props.time};
    this._update = this._updateTime.bind(this);
  }

  render() {
    var time = this._formatTime(this.state.time);
    return (
      <h1>{ time.hours } : { time.minutes } : { time.seconds }</
h1>
    );
  }

  componentDidMount() {
    this._interval = setInterval(this._update, 1000);
  }

  componentWillUnmount() {
    clearInterval(this._interval);
  }

  _formatTime(time) {
    var [ hours, minutes, seconds ] = [
      time.getHours(),
      time.getMinutes(),
      time.getSeconds()
    ].map(num => num < 10 ? '0' + num : num);

    return {hours, minutes, seconds};
  }
```

```
  _updateTime() {
    this.setState({time: new Date(this.state.time.getTime() + 10
00)});
  }
}

ReactDOM.render(<Clock time={ new Date() }/>, ...);
```

解决办法.

我们将组件拆分成容器(container)组件和UI(presentation)组件

容器组件

容器组件关心数据(包括数据的格式和数据的来源等). 容器组件关心具体的业务逻辑, 它接收数据并将数据整理成我们的UI组件需要的格式传递给UI组件. 我们常使用高阶组件去建立容器组件. 一般情况下, 容器组件的render方法里面包含的只会是UI组件.

```
// Clock/index.js
import Clock from './Clock.jsx'; // <-- Clock是一个UI组件

export default class ClockContainer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {time: props.time};
    this._update = this._updateTime.bind(this);
  }

  render() {
    return <Clock { ...this._extract(this.state.time) }/>;
  }

  componentDidMount() {
    this._interval = setInterval(this._update, 1000);
  }

  componentWillUnmount() {
    clearInterval(this._interval);
  }

  _extract(time) {
    return {
      hours: time.getHours(),
      minutes: time.getMinutes(),
      seconds: time.getSeconds()
    };
  }

  _updateTime() {
    this.setState({time: new Date(this.state.time.getTime() + 10
00)});
  }
};
```

UI组件

UI组件关心组件展示出来是什么样子. UI组件一般由基本的html标签为基础, 用以在页面上展示. 理想的UI组件应该被设计为没有外部依赖的组件. 常用的实现是使用没有内部state的无状态组件(stateless function)

```
// Clock/Clock.jsx
export default function Clock(props) {
  var [ hours, minutes, seconds ] = [
    props.hours,
    props.minutes,
    props.seconds
  ].map(num => num < 10 ? '0' + num : num);

  return <h1>{ hours } : { minutes } : { seconds }</h1>;
};
```

容器组件封装了封装了业务逻辑, 并且可以灵活的讲不同的数据注入不同的UI组件中, 这是使用容器组件带来明显的好处. 常见使用容器组件的方法是我们不去在容器组件内部规定哪个UI组件将被渲染, 而是建立一个接收一个UI组件的函数, 去灵活的让我们的容器组件可以包裹任意UI组件. 比如, 和使用下面的形式相比:

```
import Clock from './Clock.jsx';
export default class ClockContainer extends React.Component {
  render() {
    return <Clock />;
  }
}
```

我们更推荐下面这种写法:

```
export default function (Component) {
  return class Container extends React.Component {
    render() {
      return <Component />;
    }
  }
}
```

使用这种方法,我们的容器就编程了能渲染任意UI组件的容器,非常的灵活. 回到时钟的例子, 如果这时候你想把数显时钟变成一个指针时钟, 只需要替换容器组件内部的UI时钟组件就可以了.

参考资料:

- https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0#.mbglcakmp
- https://github.com/krasimir/react-in-patterns/tree/master/patterns/presentational-and-container
- https://medium.com/@learnreact/container-components-c0e67432e005

三方集成

下面这篇教程会简单介绍React如何和三方库集成.

在这个例子中我们将会学习到如何混合使用React和jQuery UI 这个插件. 我们选用了tag-it这个jQuery插件来举例. 这个插件讲无序列表(unordered list)转化成input标签来管理.

```
<ul>
  <li>JavaScript</li>
  <li>CSS</li>
</ul>
```

为了让上面这段代码能够运行, 我们需要引入jQuery, jQuery UI 以及tag-it这个插件. 使用插件的代码如下.

```
$('<dom element selector>').tagit();
```

我们选择了一个DOM元素然后调用了tagit方法.

首先我们要做的事情是对Tags这个组件我们需要它只被React渲染一次(single-renderer). 这是因为当React渲染出我们想控制的DOM元素之后,我们想把该元素的控制权从React转交给jQuery. 如果我们跳过了这一步,那么React和jQuery会对相同的DOM元素进行控制, 并且不会知道彼此的存在. 为了实现这个一次渲染的机制, 我们需要用到React自带的生命周期方法 `shouldComponentUpdate` .

当我们想以编程的方式在已有的tag-itDOM对象上添加新的标签时, 这一行为将被这个React组件触发, 并且需要配合上jQuery API一起才能工作. 我们需要找到一种方法, 既能让数据和Tags组件交互,又能保证组件只渲染一次. 为了更形象的描述我们的实现过程, 我们会在我们的APP组件里面添加一个input和一个button. 当button被点击时, 我们会讲一个string传递到Tags组件中.

```
class App extends React.Component {
  constructor(props) {
    super(props);

    this._addNewTag = this._addNewTag.bind(this);
    this.state = {
      tags: ['JavaScript', 'CSS'],
      newTag: null
    };
  }

  _addNewTag() {
    this.setState({newTag: this.refs.field.value});
  }

  render() {
    return (
      <div>
        <p>Add new tag:</p>
        <div>
          <input type='text' ref='field'/>
          <button onClick={ this._addNewTag }>Add</button>
        </div>
        <Tags tags={ this.state.tags } newTag={ this.state.newTa
g }/>
      </div>
    );
  }
}
```

我们使用了组件内部的state来存储我们新加入的field. 当我们每一次点击button的时候, state都会被更新从而触发Tags组件的重新渲染. 然而, 因为在shouldComponentUpdate中我们返回了false, 所以组件事实上并不会被更新. 还有另外一点不同的是我们通过另一个生命周期方法componentWillReceiveProps取到了新标签的值, 同时调用tagit方法来增加我们的filed.

```
class Tags extends React.Component {
  componentDidMount() {
    this.list = $(this.refs.list);
    this.list.tagit();
  }

  shouldComponentUpdate() {
    return false;
  }

  componentWillReceiveProps(newProps) {
    this.list.tagit('createTag', newProps.newTag);
  }

  render() {
    return (
      <ul ref='list'>
        { this.props.tags.map((tag, i) => <li key={ i }>{ tag }
</li>) }
      </ul>
    );
  }
}
```

参考资料:

- https://github.com/krasimir/react-in-patterns/tree/master/patterns/third-party

给setState传入回调函数

在async-nature-of-setState中我们已经提到过, setState其实是异步的. 因为出于性能优化考虑, React会将多次setState做一次批处理. 于是setState并不会在被调用之后立即改变我们的state. 这就意味着你并不能依赖于在调用setState方法之后state, 因为此时你并不能确认该state更新与否. 当然针对这个问题我们也有解决办法--用前一个state(previous state)作为需要传入函数的参数,将一个函数作为第二个参数传递给setState. 这样做能保证你传入的函数需要取到的state一定会是被传入的setState执行之后的state.

问题

```
// assuming this.state.count === 0
this.setState({count: this.state.count + 1});
this.setState({count: this.state.count + 1});
this.setState({count: this.state.count + 1});
// this.state.count === 1, not 3
```

解决办法

```
this.setState((prevState, props) => ({
  count: prevState.count + props.increment
}));
```

举一反三

```
// Passing object
this.setState({ expanded: !this.state.expanded });

// Passing function
this.setState(prevState => ({ expanded: !prevState.expanded }));
```

参考资料:

- setState() Gate
- Do I need to use setState(function) overload in this case?
- Functional setState is the future of React

装饰器

装饰器(Decorators)(被*babel*支持, 在 *03/17* 之后作为*stage-2*的*proposal*被引入)

如果你在使用类似于mobx的库, 你能够使用装饰器装饰你的函数. 装饰器本质上其实就是将组件传入一个函数. 使用装饰器能让组件更灵活,更可读并且更易修改组件的功能.

不使用装饰器的例子

```
class ProfileContainer extends Component {
  // Component code
}
export default observer(ProfileContainer)
```

使用装饰器的例子

```
@observer
export default class ProfileContainer extends Component {
  // Component code
}
```

相关文章:

- Enhancing React components with Decorators

参考资料:

- Decorators != higher ordered components
- React Decorator example - Module
- What is the use of Connect(decorator in react-redux)
- Decorators with React Components
- Exploring ES7 decorators
- Understanding Decorators

功能开关

使用Redux在React中实现Feature标记符

```
// createFeatureFlaggedContainer.js
import React from 'react';
import { connect } from 'react-redux';
import { isFeatureEnabled } from './reducers'

export default function createFeatureFlaggedContainer({
  featureName,
  enabledComponent,
  disabledComponent
  }) {
  function FeatureFlaggedContainer({ isEnabled, ...props }) {
    const Component = isEnabled ? enabledComponent : disabledCom
ponent;

    if (Component) {
      return <Component {...props} />;
    }

    // `disabledComponent` is optional property
    return null;
  }

  // Having `displayName` is very useful for debugging.
  FeatureFlaggedContainer.displayName = `FeatureFlaggedContainer
(${ featureName })`;

  return connect((store) => {
    isEnabled: isFeatureEnabled(store, featureName)
  })(FeatureFlaggedContainer);
}
```

```
// EnabledFeature.js
import { connect } from 'react-redux';
import { isFeatureEnabled } from './reducers'

function EnabledFeature({ isEnabled, children }) {
  if (isEnabled) {
    return children;
  }

  return null;
}

export default connect((store, { name }) => {
  isEnabled: isFeatureEnabled(store, name)
})(EnabledFeature);
```

```
// featureEnabled.js
import createFeatureFlaggedContainer from './createFeatureFlagge
dContainer'

// Decorator for "Page" components.
// usage: enabledFeature('unicorns')(UnicornsPage);
export default function enabledFeature(featureName) {
  return (Component) => {
    return createFeatureFlaggedContainer({
      featureName,
      enabledComponent: Component,
      disabledComponent: PageNotFound, // 404 page or something
similar
    });
  };
};
```

```
// features.js
// This is quite simple reducer, containing only an array of fea
tures.
// You can attach this data to a `currentUser` or similar reduce
r.

// `BOOTSTAP` is global action, which contains the initial data
for a page
// Features access usually don't change during user usage of a p
age
const BOOTSTAP = 'features/receive';

export default function featuresReducer(state, { type, payload }
) {
  if (type === BOOTSTAP) {
    return payload.features || [];
  }

  return state || [];
}

export function isFeatureEnabled(features, featureName) {
  return features.indexOf(featureName) !== -1;
}
```

```
// reducers.js
// This is your main reducer.js file
import { combineReducers } from 'redux';

import features, { isFeatureEnabled as isFeatureEnabledSelector
} from './features';
// ...other reducers

export default combineReducers({
  features
  // ...other reducers
});

// This is the important part, access to `features` reducer shou
ld only happens
// via this selector.
// Then you can always change where/how the features are stored.
export function isFeatureEnabled({ features }, featureName) {
  return isFeatureEnabledSelector(features, featureName);
}
```

参考资料:

- Feature flags in React
- Gist

组件切换

一个可切换的组件实际上是包含了多个组件, 选择渲染其中某个组件的组件. 我们使用对象来将props的值和组件做上映射.

```
import HomePage from './HomePage.jsx';
import AboutPage from './AboutPage.jsx';
import UserPage from './UserPage.jsx';
import FourOhFourPage from './FourOhFourPage.jsx';

const PAGES = {
  home: HomePage,
  about: AboutPage,
  user: UserPage
};

const Page = (props) => {
  const Handler = PAGES[props.page] || FourOhFourPage;

  return <Handler {...props} />
};

// The keys of the PAGES object can be used in the prop types to
 catch dev-time errors.
Page.propTypes = {
  page: PropTypes.oneOf(Object.keys(PAGES)).isRequired
};
```

参考资料:

- https://hackernoon.com/10-react-mini-patterns-c1da92f068c5

深入某个组件内部

通过父组件去访问子组件. 比如一个能自动focus的输入框(通过父组件控制自动focus)

子组件

子组件是一个带有input标签和focus方法的组件. 其中focus方法能focus到对应的HTML元素上.

```
class Input extends Component {
  focus() {
    this.el.focus();
  }

  render() {
    return (
      <input
        ref={el=> { this.el = el; }}
      />
    );
  }
}
```

父组件

在父组件中,我们能得到子组件的引用并且调用子组件的focus方法.

```
class SignInModal extends Component {
  componentDidMount() {
    // Note that when you use ref on a component, it's a referen
ce to
    // the component (not the underlying element), so you have a
ccess to its methods.
    this.InputComponent.focus();
  }

  render() {
    return (
      <div>
        <label>User name:</label>
        <Input
          ref={comp => { this.InputComponent = comp; }}
        />
      </div>
    )
  }
}
```

参考资料:

- https://hackernoon.com/10-react-mini-patterns-c1da92f068c5

集合组件

列表和其他类型的集合某种程度上也可以用组件来描述.

为了避免完全给列表新建一个单独的组件, 我们可以使用以下这种写法.

```
const SearchSuggestions = (props) => {
  // renderSearchSuggestion() behaves as a pseudo SearchSuggesti
on component
  // keep it self contained and it should be easy to extract lat
er if needed
  const renderSearchSuggestion = listItem => (
    <li key={listItem.id}>{listItem.name} {listItem.id}</li>
  );

  return (
    <ul>
      {props.listItems.map(renderSearchSuggestion)}
    </ul>
  );
};
```

如果你想在更复杂的场景里面或者其他什么地方使用这个组件, 你能很轻松的复制这段代码到新的组件中. (不要过度设计组件) If things get more complex or you want to use this component elsewhere, you should be able to copy/paste the code out into a new component. Don’t prematurely componentize.

参考资料:

- https://hackernoon.com/10-react-mini-patterns-c1da92f068c5

用来做format的组件

在我们需要格式化字符串的时候，我们可以在render方法中去使用helper函数。然而，更好的方法是去使用一个组件去做这件事。

使用组件的做法

render函数的实现会更加的干净因为只会是一些组件的组合。

```
const Price = (props) => {
  // toLocaleString 并不是React的API而是原生JavaScript的内置方法
  // https://developer.mozilla.org/en/docs/Web/JavaScript/Refere
nce/Global_Objects/Number/toLocaleString
  const price = props.children.toLocaleString('en', {
    style: props.showSymbol ? 'currency' : undefined,
    currency: props.showSymbol ? 'USD' : undefined,
    maximumFractionDigits: props.showDecimals ? 2 : 0
  });

  return <span className={props.className}>{price}</span>
};

Price.propTypes = {
  className: PropTypes.string,
  children: PropTypes.number,
  showDecimals: PropTypes.bool,
  showSymbol: PropTypes.bool
};

Price.defaultProps = {
  children: 0,
  showDecimals: true,
  showSymbol: true,
};

const Page = () => {
  const lambPrice = 1234.567;
  const jetPrice = 999999.99;
```

```
  const bootPrice = 34.567;

  return (
    <div>
      <p>One lamb is <Price className="expensive">{lambPrice}</Price></p>
      <p>One jet is <Price showDecimals={false}>{jetPrice}</Price></p>
      <p>Those gumboots will set ya back
        <Price
        showDecimals={false}
        showSymbol={false}>
        {bootPrice}
        </Price>
        bucks.
      </p>
    </div>
  );
};
```

不使用组件的做法.

代码量更少, 但是让render方法看起来不那么干净(作者个人感觉, 哈哈)

```
function numberToPrice(num, options = {}) {
  const showSymbol = options.showSymbol !== false;
  const showDecimals = options.showDecimals !== false;

  return num.toLocaleString('en', {
    style: showSymbol ? 'currency' : undefined,
    currency: showSymbol ? 'USD' : undefined,
    maximumFractionDigits: showDecimals ? 2 : 0
  });
}

const Page = () => {
  const lambPrice = 1234.567;
  const jetPrice = 999999.99;
  const bootPrice = 34.567;

  return (
    <div>
      <p>One lamb is <span className="expensive">{numberToPrice(
lambPrice)}</span></p>
      <p>One jet is {numberToPrice(jetPrice, { showDecimals: fal
se })}</p>
      <p>Those gumboots will set ya back
        {numberToPrice(bootPrice, { showDecimals: false, showSym
bol: false })}
        bucks.</p>
    </div>
  );
};
```

参考资料:

- 10 React Mini Patterns

共享tracking的逻辑

使用高阶组件能在很多的UI组件中追踪逻辑. 例如: 在许多组件中加入分析数据的追踪.

eg. Adding analytics tracking across various components.

- 一次实现,多次使用
- 易于剔除, 我们的组件还是能保持很好的测试性,不会被这个tracking组件影响.

```
import tracker from './tracker.js';

// 高阶组件
const pageLoadTracking = (ComposedComponent) => class HOC extends
 Component {
  componentDidMount() {
    tracker.trackPageLoad(this.props.trackingData);
  }

  componentDidUpdate() {
    tracker.trackPageLoad(this.props.trackingData);
  }

  render() {
    return <ComposedComponent {...this.props} />
  }
};

// 用法
import LoginComponent from "./login";

const LoginWithTracking = pageLoadTracking(LoginComponent);

class SampleComponent extends Component {
  render() {
    const trackingData = {/** Nested Object **/};
    return <LoginWithTracking trackingData={trackingData}/>
  }
}
```

坏实践

熟悉常见的坏实践能帮助我们理解React是如何工作的并且给我们重构代码提供不错的指导.

根据props去初始化state

前言:

> 使用props去在getInitialState中生成初始state(或者在constructor中初始化)很容易导致多个数据源的问题, 也会给使用者带来这样的疑问: 我们的真正的数据源到底来自哪? 这是因为getInitialState只在组件第一次初始化的时候被调用一次.

这样做的危险在于, 有可能组件的props发生了改变但是组件却没有被更新.(见下面的例子) 新的props的值不会被React认为是更新的数据因为构造器(constructor)或者getInitialState方法在组件创建之后不会再次被调用了,因此组件的state不再会被更新. 要记住, State的初始化只会在组件第一次初始化的时候发生.

坏的实践

```
class SampleComponent extends Component {
  // constructor function (or getInitialState)
  constructor(props) {
    super(props);
    this.state = {
      flag: false,
      inputVal: props.inputValue
    };
  }

  render() {
    return <div>{this.state.inputVal && <AnotherComponent/>}</div
>
  }
}
```

好的实践

```
class SampleComponent extends Component {
  // constructor function (or getInitialState)
  constructor(props) {
    super(props);
    this.state = {
      flag: false
    };
  }

  render() {
    return <div>{this.props.inputValue && <AnotherComponent/>}</
div>
  }
}
```

参考资料:

- React Anti-Patterns: Props in Initial State
- Why is passing the component initial state a prop an anti-pattern?

使用refs而不是findDOMNode()去获取DOM节点.

注意: React实际上也支持使用字符串作为ref, 来访问DOM节点. 但是需要注意的是这是一种已经不被官方推荐的用法.

- 更多关于ref的知识
- 为什么字符串形式的ref已经不被推荐了?

使用**this**找到**DOM**节点

以前的做法:

```
class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this).scrollIntoView();
  }

  render() {
    return <div />
  }
}
```

使用**ref**之后的做法

```
class MyComponent extends Component {
  componentDidMount() {
    this.node.scrollIntoView();
  }

  render() {
    return <div ref={node => this.node = node}/>
  }
}
```

使用字符串**ref**找到**DOM**节点

使用字符串**ref**的做法

```
class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this.refs.something).scrollIntoView();
  }

  render() {
    return (
      <div>
        <div ref='something'/>
      </div>
    )
  }
}
```

使用回调**ref**的做法

```
class MyComponent extends Component {
  componentDidMount() {
    this.something.scrollIntoView();
  }

  render() {
    return (
      <div>
        <div ref={node => this.something = node}/>
      </div>
    )
  }
}
```

调用子组件的**ref**

不使用**ref**的做法**:**

```
class Field extends Component {
  render() {
    return <input type='text'/>
  }
}

class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this.refs.myInput).focus();
  }

  render() {
    return (
      <div>
        Hello,
        <Field ref='myInput'/>
      </div>
    )
  }
}
```

使用**ref**的做法

```
class Field extends Component {
  render() {
    return (
      <input type='text' ref={this.props.inputRef}/>
    )
  }
}

class MyComponent extends Component {
  componentDidMount() {
    this.inputNode.focus();
  }

  render() {
    return (
      <div>
        Hello,
        <Field inputRef={node => this.inputNode = node}/>
      </div>
    )
  }
}
```

参考资料:

- ESLint Rule proposal: warn against using findDOMNode()
- Refs and the DOM

请使用高阶组件而不是Mixin

简单的例子

```
// 使用mixin
var WithLink = React.createClass({
  mixins: [React.addons.LinkedStateMixin],
  getInitialState: function () {
    return {message: 'Hello!'};
  },
  render: function () {
    return <input type="text" valueLink={this.linkState('message
')}/>;
  }
});

// 使用高阶组件的做法
var WithLink = React.createClass({
  getInitialState: function () {
    return {message: 'Hello!'};
  },
  render: function () {
    return <input type="text" valueLink={LinkState(this,'message
')}/>;
  }
});
```

更加详细的例子

```
// 使用Mixin Mixin
var CarDataMixin = {
  componentDidMount: {
    // fetch car data and
    // call this.setState({carData: fetchedData}),
    // once data has been (asynchronously) fetched
  }
```

```
};

var FirstView = React.createClass({
  mixins: [CarDataMixin],
  render: function () {
    return (
      <div>
        <AvgSellingPricesByYear country="US" dataset={this.state
.carData}/>
        <AvgSellingPricesByYear country="UK" dataset={this.state
.carData}/>
        <AvgSellingPricesByYear country="FI" dataset={this.state
.carData}/>
      </div>
    )
  }
});

// 使用高阶组件
var bindToCarData = function (Component) {
  return React.createClass({
    componentDidMount: {
      // fetch car data and
      // call this.setState({carData: fetchedData}),
      // once data has been (asynchronously) fetched
    },

    render: function () {
      return <Component carData={ this.state.carData }/>
    }
  });
};

// 将你的组件使用高阶组件包裹起来
var FirstView = bindToCarData(React.createClass({
  render: function () {
    return (
      <div>
        <AvgSellingPricesByYear country="US" dataset={this.props
.carData}/>
```

```
        <AvgSellingPricesByYear country="UK" dataset={this.props
.carData}/>
        <AvgSellingPricesByYear country="FI" dataset={this.props
.carData}/>
      </div>
    )
  }
}));
```

参考资料:

- Mixins are dead - Long live higher ordercomponents
- Mixins are considered harmful
- Stackoverflow: Using mixins vs components for code reuse
- Stackoverflow: Composition instead of mixins in React

避免在 `componentWillMount()` 中使用进行 `setState` 操作.

`componentWillMount()` 在组件将要挂载时被立即调用. 这个调用发生在 `render()` 函数执行之前, 所以如果在 `componentWillMount` 里面设置了state, 这个设置的state是不会触发重新渲染的. 同样我们也需要注意不要在 `componentWillMount()` 中引入其他可能会导致问题的代码.

如果你有类似的需求, 请在 `componentDidMount` 里面完成.

```
function componentDidMount() {
  axios.get(`api/messages`)
    .then((result) => {
      const messages = result.data
      console.log("COMPONENT WILL Mount messages : ", messages);
      this.setState({
        messages: [...messages.content]
      })
    })
}
```

不使用setState() 去操作state

导致的问题

- 在state改变时组件不会重新渲染.
- 在未来某个时候如果通过setState改变了state, 那么这次未通过setState去改变的state将会同样生效.

坏实践

```
class SampleComponent extends Component {
  constructor(props) {
    super(props);

    this.state = {
      items: ['foo', 'bar']
    };

    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // 坏实践：我们手动更改了state而不是通过setState函数.
    this.state.items.push('lorem');

    this.setState({
      items: this.state.items
    });
  }

  render() {
    return (
      <div>
        {this.state.items.length}
        <button onClick={this.handleClick}>+</button>
      </div>
    )
  }
}
```

好实践

```
class SampleComponent extends Component {
  constructor(props) {
    super(props);

    this.state = {
      items: ['foo', 'bar']
    };

    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // 我们使用了setState()方法来更新state, 组件将会在state更改后被更新.

    this.setState({
      items: this.state.items.concat('lorem')
    });
  }

  render() {
    return (
      <div>
        {this.state.items.length}
        <button onClick={this.handleClick}>+</button>
      </div>
    )
  }
}
```

参考资料:

React Design Patterns and best practices by Michele Bertoli.

使用简单索引作为key

Keys 应该是稳定,可预测,并且唯一的. 这样React才能正确追踪到某一个元素.

坏实践

在下面这段代码中,每个元素的key事实上是它在todos这个数组里面的顺序, 而事实上更好的实践应该是把key和我们想要表达的数据紧紧关联在一起. 下面这种做法会阻碍React对于我们组件的优化.

```
{todos.map((todo, index) =>
  <Todo
    {...todo}
    key={index}
  />
)}
```

好实践

假设 `todo.id` 是唯一的并且稳定的, React便能更好的去控制这些组件的更新(否则我们可能会面临大量重复创建的组件, 并且每次更新都是重新render这些组件.)

```
{todos.map((todo) =>
  <Todo {...todo}
    key={todo.id} />
)}
```

参考资料:

- React docs
- Lin Clark's code cartoon

将props展平传到DOM上

当我们将展平(spread)的props传入子组件时我们便引入了风险，因为我们可能往HTML标签上添加了它并不支持的属性。

坏实践

下面这个例子会在DOM元素上增加一个该元素本身并不支持的属性 `flag` 。

```
const Sample = () => (<Spread flag={true} className="content"/>)
;
const Spread = (props) => (<div {...props}>Test</div>);
```

好实践

如果将HTML DOM元素需要接受的props分离出来再展开传入，会是一种更安全的做法。

```
const Sample = () => (<Spread flag={true} domProps={{className: "
content"}}/>);
const Spread = (props) => (<div {...props.domProps}>Test</div>);
```

或者我们也可以使用 `...rest` 去过滤掉那些HTML DOM并不支持的属性。

```
const Sample = () => (<Spread flag={true} className="content"/>)
;
const Spread = ({ flag, ...domProps }) => (<div {...domProps}>Te
st</div>);
```

需要注意的是

在这种情况下，当我们使用了PureComponent时，即使 `domProps` 没有变化时，组件还是会被重新渲染。因为PureComponent对于对象使用的是浅比较

参考资料:

- React Design Patterns and best practices by Michele Bertoli.
- In React, children are just props: Kent C. Dodds' Tweet

Handling UX variations for multiple brands and apps

Vasa做的关于react组件复用的演讲

以下是一些有助于编写高复用性react组件的通用编码原则。

单功能原则

使用**react**时

组件或容器的代码在根本上必须只负责一块UI功能。

- 以设计收货地址组件为例
- 收货地址组件可以拆分为地址组件（只含有地址相关内容），姓名组件（包括姓氏和名字），电话组件，省，市和邮政编码组件。

使用**redux**时

All API related call go into Redux thunks/other async handling sections (redux-promise, sagas etc)所有API相关操作可以使用redux或者其他异步模块（如redux-promise, sagas等）来处理。模块可以如下划分：

- 一些模块只负责在AJAX请求成功或失败的时候派发动作。
- 一些模块通过promise来接收。

让组件保持简单 (KISS)

- 如果组件根本不需要状态，那么就使用函数定义的无状态组件。
- 从性能上来说，函数定义的无状态组件 **> ES6 class** 定义的组件 **>** 通过 **React.createClass()**定义的组件。
- 仅传递组件所需要的属性。只有当属性列表太长时，才使用{...this.props}进行传递。

- 如果组件里面有太多的判断逻辑（if-else语句）通常意味着这个组件需要被拆分成更细的组件或模块。
- 还有一点，使用明确的命名能够让开发者明白它的功能，有助于组件复用。

相关文章

Building React Components for Multiple Brands and Applications

使用组合去实现不同的交互功能

在React的实践中, 将小的可复用的组件组合成功能更复杂的组件是一种推荐的实践.

我们怎么保证组件的复用性**?**

- 我们需要保证组件是纯的UI组件, 传入同样的props总会渲染出相同的组件.(木偶组件)

复用意味着什么**?**

- 在组件内部没有与外部的数据交互(这应该在Redux中完成).
- 如果有需要从API获取的数据, 请使用redux-thunk. redux-thunk和redux容器是相互隔离的, 我们可以通过redux-thunk获取数据, 然后通过redux容器将数据以props的形式传递到我们的子组件里面去.

如果我们在 `render` 函数里面使用了很多 `renderxxx()` 该怎么办?

- 如果使用或者创建了很多renderxxx()函数, 这往往意味着这些renderxxx()函数是可以变成可复用的小组件的.

例子

千变万化的登录页面

用户登录页面可能变化很多, 根据用户登录情况会打开/关闭某些功能或者显示/隐藏一些元素.

这些打开/关闭的功能应该被封装在一个子组件里面, 而一些无论什么情况下都要显示的元素(header/footer), 我们可以抽取出来放在父组件中进行复用.

```
import React, { Component } from "react";
import PropTypes from 'prop-types';
import SignIn from "./sign-in";

class MemberSignIn extends Component {
  _renderMemberJoinLinks() {
    return (
      <div className="member-signup-links">
        ...
      </div>
    );
  }

  _routeTo() {
    // Routing logic here
  }

  render() {
    const {forgotEmailRoute,forgotPwdRoute, showMemberSignupLink
s} = this.props;
    return (
      <div>
        <SignIn
          onForgotPasswordRequested={this._routeTo(forgotPwdRout
e)}
          onForgotEmailRequested={this._routeTo(forgotEmailRoute
)}>
          {this.props.children}
          {showMemberSignupLinks && this._renderMemberJoinLinks(
)}
        </SignIn>
      </div>
    );
  }
}

export default MemberSignIn;
```

参考资料:

- Slides from my talk: Building Multi-tenant UI with React

样式开关

我们也可以通过开/关某个功能来实现我们我们组件的多样性. 而这些开关我们可以通过props传递进我们的组件.

Toggle并不是万能的:

如果抱着toggle的想法, 我们很容易将一个组件设计成一个万能组件, 通过传入一大堆props来完成多样性, 这其实并不是特别好的实践. 所以我们最好遵守以下两点:

- 只将必须的props传入我们的组件, 而且当props太多时需要考虑抽取子组件.
- 不要违背单一职责原则.

例子

在登录表单中显示/隐藏 password

```
class PasswordField extends Component {
  render() {
    const {
      password,
      showHidePassword,
      showErrorOnTop,
      showLabels,
      shouldComplyAda
    } = this.props;
    return (
      <div>
        <Password
          field={password}
          label="Password"
          showErrorOnTop={showErrorOnTop}
          placeholder={shouldComplyAda ? "" : "Password"}
          showLabel={showLabels}
          showHidePassword={showHidePassword}
        />
      </div>
    );
  }
}
```

参考资料:

- Slides from my talk: Building Multi-tenant UI with React

用高阶组件去实现功能开关

使用高阶组件去实现我们的toggle, 从而实现组件的多样性.

比如下面的例子, 实现某个功能的开/关

```
// featureToggle.js
const isFeatureOn = function (featureName) {
  // return true or false
};

import { isFeatureOn } from './featureToggle';

const toggleOn = (featureName, ComposedComponent) => class HOC extends Component {
  render() {
    return isFeatureOn(featureName) ? <ComposedComponent {...this.props} /> : null;
  }
};

// 用法
import AdsComponent from './Ads'
const Ads = toggleOn('ads', AdsComponent);
```

使用高阶组件做props代理.

Props代理

使用高阶组件能帮助我们对于传入的props进行修饰后传入真正的组件(类似于middleware的概念)

```
function HOC(WrappedComponent) {
  return class Test extends Component {
    render() {
      const newProps = {
        title: 'New Header',
        footer: false,
        showFeatureX: false,
        showFeatureY: true
      };

      return <WrappedComponent {...this.props} {...newProps} />
    }
  }
}
```

Wrapper Components

对我们的组件进行包装来适配不同的样式/交互行为. 如果你想处理 `<div>` 或者其他HTML标签的话, 你可以使用组合.

当你创建React实例的时候, 你能在jsx标签内包裹其他的React组件或者任意的JavaScript表达式. 父组件通过this.props.children能访问到其包裹的子组件.

```
const SampleComponent = () => {
  <Parent>
    <Child />
  </Parent>
};

const Parent = () => {
  // 你能使用class 'bla'或者其他的class来给子组件加上不同的样式.
  <div className="bla">
    {this.props.children}
  </div>
};
```

值得一提的是, 包裹组件同样可以通过接收一个tag名来生成对应的HTML标签. 但是一般情况下我们不推荐这么做, 因为这样做的话你就不能添加属性或者传入props了.

```
const SampleComponent = () => {
  <Wrap tagName="div" content="Hello World" />
};

const Wrap = ({ tagName, content }) => {
  const Tag = `${tagName}`     // 变量名必须大写开头因为这是一个组件.
  return <Tag>{content}</Tag>
}
```

参考资料:

- Slides from my talk: Building Multi-tenant UI with React

以不同的顺序展示我们的UI组件

我们使用props来定下我们显示的顺序. 我们的组件基于我们排好序的props进行渲染.

```
class PageSections extends Component {
  render() {
    const pageItems = this.props.contentOrder.map(
      (content) => {
        const renderFunc = this.contentOrderMap[content];
        return (typeof renderFunc === 'function') ? renderFunc()
 : null;
      }
    );

    return (
      <div className="page-content">
        {pageItems}
      </div>
    )
  }
}
```

参考资料:

- Slides from my talk: Building Multi-tenant UI with React

Perf Tips

基本准则

- 在 `shouldComponentUpdate` 中避免不必要的检查.
- 使用不可变数据类型(Immutable).
- 编写针对产品环境的打包配置(Production Build).
- 通过Chrome Timeline来记录组件所耗费的资源.
- 在 `componentWillMount` 或者 `componentDidMount` 里面通过 `setTimeOut` 或者 `requestAnimationFram` 来延迟执行那些需要大量计算的任务.

相关文章

Optimizing Performance: Docs

Performance Engineering with React

Tips to optimise rendering of a set of elements in React

React.js Best Practices for 2016

shouldComponentUpdate检查

合理的实现 `shouldComponentUpdate` 能够避免不必要的重新渲染.

React会在props和state发生改变的时候重新渲染组件. 试想一下当每次有props和state发生改变时(可能只是一个很小的用户动作), 整个页面都会重新渲染一次, 这从性能上来说肯定不能令人满意. 这就是 `shouldComponentUpdate` 这个生命周期函数发挥作用的时候了. 当React想要重新渲染组件时, React会检查 `shouldComponentUpdate` 这个函数是返回true还是false(这将决定组件是否更新.)React默认这个函数是返回true的,意味着无论state还是props发生变化, 组件都将被更新). 所以对于那些不需要变化的组件, 我们可以直接返回false来阻止组件更新,以此提升性能. 但更多的时候, 我们需要在这个函数内写自己的逻辑来判断组件是否需要更新.

坏实践

```
const AutocompleteItem = (props) => {
  const selectedClass = props.selected === true ? "selected" : ""
;
  var path = parseUri(props.url).path;
  path = path.length <= 0 ? props.url : "..." + path;

  return (
    <li
      onMouseLeave={props.onMouseLeave}
      className={selectedClass}>
      <i className="ion-ios-eye"
         data-image={props.image}
         data-url={props.url}
         data-title={props.title}
         onClick={props.handlePlanetViewClick}/>
      <span
        onMouseEnter={props.onMouseEnter}
      >
        <div className="dot bg-mint"/>
        {path}
      </span>
    </li>
  );
};
```

好实践

```
export default class AutocompleteItem extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    if (
      nextProps.url !== this.props.url ||
      nextProps.selected !== this.props.selected
    ) {
      return true;
    }
    return false;
  }

  render() {
    const {props} = this;
    const selectedClass = props.selected === true ? "selected" :
"";
    var path = parseUri(props.url).path;
    path = path.length <= 0 ? props.url : "..." + path;

    return (
      <li
        onMouseLeave={props.onMouseLeave}
        className={selectedClass}>
        <i className="ion-ios-eye"
           data-image={props.image}
           data-url={props.url}
           data-title={props.title}
           onClick={props.handlePlanetViewClick}/>
        <span
          onMouseEnter={props.onMouseEnter}>
          <div className="dot bg-mint"/>
          {path}
        </span>
      </li>
    );
  }
}
```

参考资料:

- React Performance optimization
- React rendering misconception

使用Pure Component

Pure Component默认会在 `shouldComponentUpdate` 方法中做浅比较. 这种实现可以避免可以避免发生在state或者props没有真正改变的重新渲染.

Recompose提供了一个叫 `pure` 的高阶组件来实现这个功能, React在v15.3.0中正式加入了 `React.PureComponent` .

坏实践

```
export default (props, context) => {
  // ... do expensive compute on props ...
  return <SomeComponent {...props} />
}
```

好实践

```
import { pure } from 'recompose';
// This won't be called when the props DONT change
export default pure((props, context) => {
  // ... do expensive compute on props ...
  return <SomeComponent someProp={props.someProp}/>
})
```

更好的写法

```
// This is better mainly because it uses no external dependencie
s.
import { PureComponent } from 'react';

export default class Example extends PureComponent {
  // This won't re-render when the props DONT change
  render() {
    // ... do expensive compute on props ...
    return <SomeComponent someProp={props.someProp}/>
  }
}
})
```

参考资料:

- Recompose
- Higher Order Components with Functional Patterns Using Recompose
- React: PureComponent
- Pure Components
- Top 5 Recompose HOCs

使用 Reselect

在React-Redux的 connect(mapState)中使用Reselect, 这能避免频繁的重新渲染的发生.

坏实践

```
let App = ({otherData, resolution}) => (
  <div>
    <DataContainer data={otherData}/>
    <ResolutionContainer resolution={resolution}/>
  </div>
);

const doubleRes = (size) => ({
  width: size.width * 2,
  height: size.height * 2
});

App = connect(state => {
  return {
    otherData: state.otherData,
    resolution: doubleRes(state.resolution)
  }
})(App);
```

在上面的代码中, 一旦otherData发生修改, 那么DataContainer和ResolutionContainer两者都会发生重新渲染, 即使在resolution的state并没有发生改变时也是这样. 这是因为doubleRes这个函数总是会返回一个新的resolutiond对象的实体. 如果doubleRes函数通过Reselect方式编写就没有这个问题了. Reslect会记录下上一次函数调用的结果并且当再次以相同方式调用时返回相同的结果(而不是创建一个一模一样的新结果). 只有当传入的参数不同时, 才会产生新的结果.

好实践

```
import { createSelector } from 'reselect';
const doubleRes = createSelector(
  r => r.width,
  r => r.height,
  (width, height) => ({
    width: width * 2,
    height: height * 2
  })
);
```

参考资料:

- React
- Computing Derived Data: Docs

React中的样式

在这一个章节中我们会提供一些CSS In JS的实践.

如果你还不太明白为什么要CSS In Js, 作者vasan推荐你看一看下面的 talk by Vjeux

相关文章

Patterns for style composition in React

Inline style vs stylesheet performance

给无状态的纯UI组件应用样式

请保持样式远离那些离不开state的组件. 比如路由, 视图, 容器, 表单, 布局等等不应该有任何的样式或者css class出现在组件上. 相反, 这些复杂的业务组件应该有一些带有基本功能的无状态UI组件组成.

没有任何样式/classNames的Form组件, 仅有纯的组件组合而成.

```
class SampleComponent extends Component {
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <Heading children='Sign In'/>
        <Input
          name='username'
          value={username}
          onChange={this.handleChange}/>
        <Input
          type='password'
          name='password'
          value={password}
          onChange={this.handleChange}/>
        <Button
          type='submit'
          children='Sign In'/>
      </form>
    )
  }
}

// 表达组件(带样式)
const Button = ({
  ...props
  }) => {
  const sx = {
    fontFamily: 'inherit',
    fontSize: 'inherit',
    fontWeight: 'bold',
```

```
    textDecoration: 'none',
    display: 'inline-block',
    margin: 0,
    paddingTop: 8,
    paddingBottom: 8,
    paddingLeft: 16,
    paddingRight: 16,
    border: 0,
    color: 'white',
    backgroundColor: 'blue',
    WebkitAppearance: 'none',
    MozAppearance: 'none'
  }

  return (
    <button {...props} style={sx}/>
  )
}
```

样式模块(style module)

一般来说, 在组件内写死(hard code)样式应该是要被避免的. 这些有可能被不同的UI组件分享的样式应该被分开放入对应的模块中.

```
// 样式模块
export const white = '#fff';
export const black = '#111';
export const blue = '#07c';

export const colors = {
  white,
  black,
  blue
};

export const space = [
  0,
  8,
  16,
  32,
  64
];

const styles = {
  bold: 600,
  space,
  colors
};

export default styles
```

Usage

```
// button.jsx
import React from 'react'
import { bold, space, colors } from './styles'

const Button = ({
  ...props
  }) => {
  const sx = {
    fontFamily: 'inherit',
    fontSize: 'inherit',
    fontWeight: bold,
    textDecoration: 'none',
    display: 'inline-block',
    margin: 0,
    paddingTop: space[1],
    paddingBottom: space[1],
    paddingLeft: space[2],
    paddingRight: space[2],
    border: 0,
    color: colors.white,
    backgroundColor: colors.blue,
    WebkitAppearance: 'none',
    MozAppearance: 'none'
  };

  return (
    <button {...props} style={sx}/>
  )
};
```

样式函数(Style Functions)

因为在React中可以很方便的使用JavaScript, 所以我们能使用helper函数来帮我们处理样式相关的问题.

第一个例子

一个用rgba格式来创造黑色的函数.

```
const darken = (n) => `rgba(0, 0, 0, ${n})`;
darken(1 / 8); // 'rgba(0, 0, 0, 0.125)'

const shade = [
  darken(0),
  darken(1 / 8),
  darken(1 / 4),
  darken(3 / 8),
  darken(1 / 2),
  darken(5 / 8),
  darken(3 / 4),
  darken(7 / 8),
  darken(1)
];
// 现在,
// shade[4] 就是 'rgba(0, 0, 0, 0.5)'
```

第二个例子

给margin 和 padding创建一个比例来保持视觉节奏的一致.

```
// Modular powers of two scale
const scale = [
  0,
  8,
  16,
  32,
  64
```

```
];

// 通过这个函数去取得一部分的样式
const createScaledPropertyGetter = (scale) => (prop) => (x) => {
  return (typeof x === 'number' && typeof scale[x] === 'number')
    ? {[prop]: scale[x]}
    : null
};
const getScaledProperty = createScaledPropertyGetter(scale);

export const getMargin = getScaledProperty('margin');
export const getPadding = getScaledProperty('padding');
// 样式函数的用法
const Box = ({
  m,
  p,
  ...props
  }) => {
  const sx = {
    ...getMargin(m),
    ...getPadding(p)
  };

  return <div {...props} style={sx}/>
};

// 组件用法.
const Box = () => (
  <div>
    <Box m={2} p={3}>
      A box with 16px margin and 32px padding
    </Box>
  </div>
);
```

使用npm模块

对于那些比较复杂的样式/颜色转换，使用不同的npm模块有时会是比自己造轮子更好的选择.

Example

对于在CSS中的暗色梯度，我们可以使用 `chroma-js` 这个模块

```
import chroma from 'chroma-js'

const alpha = (color) => (a) => chroma(color).alpha(a).css();

const darken = alpha('#000');

const shade = [
  darken(0),
  darken(1 / 8),
  darken(1 / 4)
  // More...
];

const blueAlpha = [
  alpha(blue)(0),
  alpha(blue)(1 / 4),
  alpha(blue)(1 / 2),
  alpha(blue)(3 / 4),
  alpha(blue)(1)
];
```

基础组件(样式通过props传入的组件)

使用基础组件

在React中使用组合的思想去构建我们的UI会带来很大的灵活性, 因为我们的组件从另一个角度来看都是函数. 通过改变组件中的props进而改变组件的样式, 我们能让组件更加的可复用.

我们把color和backgroundColor属性作为组件的props传入, 另外我们新加了一个props来调整padding top和padding bottom.

```
const Button = ({
  big,
  color = colors.white,
  backgroundColor = colors.blue,
  ...props
  }) => {
  const sx = {
    fontFamily: 'inherit',
    fontSize: 'inherit',
    fontWeight: bold,
    textDecoration: 'none',
    display: 'inline-block',
    margin: 0,
    paddingTop: big ? space[2] : space[1],
    paddingBottom: big ? space[2] : space[1],
    paddingLeft: space[2],
    paddingRight: space[2],
    border: 0,
    color,
    backgroundColor,
    WebkitAppearance: 'none',
    MozAppearance: 'none'
  };

  return (
    <button {...props} style={sx}/>
  )
};
```

用法

```
const Button = () => (
  <div>
    <Button>
      Blue Button
    </Button>
    <Button big backgroundColor={colors.red}>
      Big Red Button
    </Button>
  </div>
);

// 通过Button组件的API去改变Button组件的样式,
// 我们能得到样式各异的Button.
const ButtonBig = (props) => <Button {...props} big/>;
const ButtonGreen = (props) => <Button {...props} backgroundColo
r={colors.green}/>;
const ButtonRed = (props) => <Button {...props} backgroundColor=
{colors.red}/>;
const ButtonOutline = (props) => <Button {...props} outline/>;
```

布局组件

我们拓展了基础组件的概念, 创造出了布局组件.

例子

```
const Grid = (props) => (
  <Box {...props}
    display='inline-block'
    verticalAlign='top'
    px={2}/>
);

const Half = (props) => (
  <Grid {...props}
    width={1 / 2}/>
);

const Third = (props) => (
  <Grid {...props}
    width={1 / 3}/>
);

const Quarter = (props) => (
  <Grid {...props}
    width={1 / 4}/>
);

const Flex = (props) => (
  <Box {...props}
    display='flex'/>
);

const FlexAuto = (props) => (
  <Box {...props}
    flex='1 1 auto'/>
);
```

用法

```
const Layout = () => (
  <div>
    <div>
      <Half>Half width column</Half>
      <Half>Half width column</Half>
    </div>
    <div>
      <Third>Third width column</Third>
      <Third>Third width column</Third>
      <Third>Third width column</Third>
    </div>
    <div>
      <Quarter>Quarter width column</Quarter>
      <Quarter>Quarter width column</Quarter>
      <Quarter>Quarter width column</Quarter>
      <Quarter>Quarter width column</Quarter>
    </div>
  </div>
);
```

参考资料:

- Github: React Layout components
- Leveling Up With React: Container Components
- Container Components and Stateless Functional Components in React

排版组件

我们拓展了基础组件的概念创造了排版组件. 这个模式能保证一致性以及你的样式足够的纯净.

例子

```
import React from 'react';
import { alternateFont, typeScale, boldFontWeight } from './styl
es';

const Text = ({
  tag = 'span',
  size = 4,
  alt,
  center,
  bold,
  caps,
  ...props
  }) => {
  const Tag = tag;
  const sx = {
    fontFamily: alt ? alternateFont : null,
    fontSize: typeScale[size],
    fontWeight: bold ? boldFontWeight : null,
    textAlign: center ? 'center' : null,
    textTransform: caps ? 'uppercase' : null
  };

  return <Tag {...props} style={sx}/>
};

const LeadText = (props) => <Text {...props} tag='p' size={3}/>;
const Caps = (props) => <Text {...props} caps/>;
const MetaText = (props) => <Text {...props} size={5} caps/>;
const AltParagraph = (props) => <Text {...props} tag='p' alt/>;

const CapsButton = ({ children, ...props }) => (
  <Button {...props}>
    <Caps>
      {children}
    </Caps>
  </Button>
);
```

用法

```
const TypographyComponent = () => (
  <div>
    <LeadText>
      This is a lead with some<Caps>all caps</Caps>.
    It has a larger font size than the default paragraph.
    </LeadText>
    <MetaText>
      This is smaller text, like form helper copy.
    </MetaText>
  </div>
);
```

使用高阶组件来改变样式

有时有一些组件可能只需要很少的一部分state来维护一些很简单的交互, 我们也有充足的理由把这些组件作为可复用的组件.

例子. Carousel组件的交互

例子: Carousel

这个高阶组件会持有当前幻灯片的index并且提供前进和回退的功能.

```
// 高阶组件
import React from 'react'
// 这个高阶组件其实可以被命名的更加通俗易懂，比如Counter或者Cycle
const CarouselContainer = (Comp) => {
  class Carousel extends React.Component {
    constructor() {
      super();
      this.state = {
        index: 0
      };
      this.previous = () => {
        const { index } = this.state;
        if (index > 0) {
          this.setState({index: index - 1})
        }
      };

      this.next = () => {
        const { index } = this.state;
        this.setState({index: index + 1})
      }
    }

    render() {
      return (
        <Comp
          {...this.props}
          {...this.state}
          previous={this.previous}
          next={this.next}/>
      )
    }
  }
  return Carousel
};
export default CarouselContainer;
```

使用高阶组件

```
// 纯UI component
const Carousel = ({ index, ...props }) => {
  const length = props.length || props.children.length || 0;

  const sx = {
    root: {
      overflow: 'hidden'
    },
    inner: {
      whiteSpace: 'nowrap',
      height: '100%',
      transition: 'transform .2s ease-out',
      transform: `translateX(${index % length * -100}%)`
    },
    child: {
      display: 'inline-block',
      verticalAlign: 'middle',
      whiteSpace: 'normal',
      outline: '1px solid red',
      width: '100%',
      height: '100%'
    }
  };

  const children = React.Children.map(props.children, (child, i)
 => {
    return (
      <div style={sx.child}>
        {child}
      </div>
    )
  });

  return (
    <div style={sx.root}>
      <div style={sx.inner}>
        {children}
      </div>
    </div>
  )
```

```
};

// 最后的Carousel组件
const HeroCarousel = (props) => {
  return (
    <div>
      <Carousel index={props.index}>
        <div>Slide one</div>
        <div>Slide two</div>
        <div>Slide three</div>
      </Carousel>
      <Button
        onClick={props.previous}
        children='Previous'/>
      <Button
        onClick={props.next}
        children='Next'/>
    </div>
  )
};

// 我们通过在外面包裹一层container组件来给这个组件带来更多的功能.
export default CarouselContainer(HeroCarousel)
```

通过保持样式和交互状态的分离，基于同一个carsousel组件，我们可以创造任意数量不同的更复杂的carousel组件，

用法

```
const Carousel = () => (
  <div>
    <HeroCarousel />
  </div>
);
```

React中的陷阱

在绝大多数情况下, React都是清晰直观的. 但是也不乏有一些小陷阱, 不注意的话有时候也会给你"意外的惊喜". 下面我们就来介绍一下这些小陷阱

参考资料

React Gotchas

Top 5 React Gotchas

保证渲染的性能

为了保障组件的性能, 我们有的时候会从组件渲染的角度出发.

更干净的render函数? 这个概念可能会有点让人疑惑.

其实在这里干净是指我们在 `shouldComponentUpdate` 这个生命周期函数里面去做浅比较, 从而避免不必要的渲染.

关于上面的干净渲染, 现有的一些实现包括React.PureComponent, PureRenderMixin, recompose/pure 等等.

第一个例子

坏实践

```
class Table extends PureComponent {
  render() {
    return (
      <div>
        {this.props.items.map(i =>
          <Cell data={i} options={this.props.options || []}/>
        )}
      </div>
    );
  }
}
```

这种写法的问题在于 **`{this.props.options || []}`** **- 这种写法会导致所有的Cell都被重新渲染即使只有一个cell发生了改变. 为什么会发生这种事呢?**

仔细观察你会发现, options这个数组被传到了Cell这个组件上, 一般情况下, 这不会导致什么问题. 因为如果有其他的Cell组件, 组件会在有props发生改变的时候浅对比props并且跳过渲染(因为对于其他Cell组件, props并没有发生改变). 但是在这个例子里面, 当options为null时, 一个默认的空数组就会被当成Props传到组件里面去. 事实上每次传入的 `[]` 都相当于创建了新的Array实例. 在JavaScript里面, 不同的实例

是有不同的实体的，所以浅比较在这种情况下总是会返回false，然后组件就会被重新渲染。因为两个实体不是同一个实体。这就完全破坏了React对于我们组件渲染的优化。

好实践

```
const defaultval = [];  // <---  也可以使用defaultProps
class Table extends PureComponent {
  render() {
    return (
      <div>
        {this.props.items.map(i =>
          <Cell data={i} options={this.props.options || defaultv
al}/>
        )}
      </div>
    );
  }
}
```

第二个例子

在render函数里面调用函数也可能导致和上面相同的问题。

坏实践

```
class App extends PureComponent {
  render() {
    return <MyInput
      onChange={e => this.props.update(e.target.value)}/>;
  }
}
```

又一个坏实践

```
class App extends PureComponent {
  update(e) {
    this.props.update(e.target.value);
  }

  render() {
    return <MyInput onChange={this.update.bind(this)}/>;
  }
}
```

在上面的两个坏实践中，每次我们都会去创建一个新的函数实体. 和第一个例子类似，新的函数实体会让我们的浅比较返回false，导致组件被重新渲染. 所以我们需要在更早的时候去bind我们的函数.

好实践

```
class App extends PureComponent {
  constructor(props) {
    super(props);
    this.update = this.update.bind(this);
  }

  update(e) {
    this.props.update(e.target.value);
  }

  render() {
    return <MyInput onChange={this.update}/>;
  }
}
```

坏实践

```
class Component extends React.Component {
  state = {clicked: false};

  onClick() {
    this.setState({clicked: true})
  }

  render() {
    // 如果options为空的话，每次都会创建一个新的{test:1}对象
    const options = this.props.options || {test: 1};

    return <Something
      options={options}
      // New function created each render
      onClick={this.onClick.bind(this)}
      // New function & closure created each render
      onTouchTap={(event) => this.onClick(event)
    />
  }
}
```

好实践

```
class Component extends React.Component {
  state = {clicked: false};
  options = {test: 1};

  onClick = () => {
    this.setState({clicked: true})
  };

  render() {
    // Options这个对象只被创建了一次.
    const options = this.props.options || this.options;

    return <Something
      options={options}
      onClick={this.onClick} // 函数只创建一次, 只绑定一次
      onTouchTap={this.onClick} // 函数只创建一次, 只绑定一次
    />
  }
}
```

参考资料:

- https://medium.com/@esamatti/react-js-pure-render-performance-anti-pattern-fb88c101332f
- https://github.com/nfour/js-structures/blob/master/guides/react-anti-patterns.md#pure-render-immutability
- Optimizing React Rendering

React中的 Synthetic 事件

React在处理事件(event时), 事实上使用了SyntheticEvent对象包裹了原生的event对象.

这些React自己维护的对象是相互联系的, 意味着如果对于某一个事件, 我们给出了对应的响应函数(handler), 其他的SyntheticEvent对象也是可以重用的.这也是React提升性能的秘诀之一. 但是这也意味着, 如果想要通过异步的方式访问事件对象是不可能的, 因为出于reuse的原因, 事件对象里面的值都被重置了.

下面这段代码会在控制台里面打出null, 因为事件在SyntheticEvent池中被重用了.

```
function handleClick(event) {
  setTimeout(function () {
    console.log(event.target.name);
  }, 1000);
}
```

为了避免这种情况, 你需要去保存你关心的事件的属性.

```
function handleClick(event) {
  let name = event.target.name;
  setTimeout(function () {
    console.log(name);
  }, 1000);
}
```

参考资料:

- React/Redux: Best practices & gotchas
- React events in depth w/ Kent C. Dodds, Ben Alpert, & Dan Abramov

Related Links

- React in Patterns by krasimir
- React Patterns by planningcenter
- reactpatterns.com
- 10 React Mini-patterns